A solution to a problem posed by S.D. Swierstra

On 1 february 1990 I presented a lecture about functional-programming techniques, among which *tupling*, at the Rijksuniversiteit of Utrecht. After this lecture, S.D. Swierstra posed me a problem with the annotation that he could only solve it by using *attribute grammars*. When, a few days later, I started to think about this problem I solved it almost immediately. Actually, most of the time I spent on it was devoted to the construction of a formal problem specification from its informal description in my memory.

The problem, as I interpreted it, can be stated as follows. For type Name we consider the type Seq defined recursively by

$$\text{Seq} \;=\; \underline{\text{list of}}\,(\,\underline{d}\ \text{Name} \mid \underline{u}\ \text{Name} \mid \underline{s}\ \text{Seq}\,)\ .$$

So, a (value of type) Seq is a list whose elements are Name's prefixed with either $\underline{d}$ or $\underline{u}$, and Seq's prefixed with $\underline{s}$. The prefixes $\underline{d}$, $\underline{u}$, and $\underline{s}$ serve to discriminate the three cases in the definition of Seq.

Elements of Seq can be interpreted as block-structured programs, in which $\underline{d}a$ represents the declaration of a variable with name a, $\underline{u}a$ represents a statement in which variable a is used, and $\underline{s}\,x$ represents an inner block x. In a machine-code representation of such a program, the "use instances" — i.e.: occurrences prefixed with $\underline{u}$ — of variables are usually replaced by, so-called, "addressing pairs": the machine-code representation of a program just is —for our purpose— a list of addressing pairs. The mapping from use instances to addressing pairs thereby depends on how the variables have been declared. Here, we confine ourselves to *correct* programs, which have the property that each variable in a use instance has been declared in some block surrounding that use instance. Otherwise, we impose no restrictions; in particular, declarations need not textually precede use instances.

<u>example</u>: [$\underline{d}$a, [$\underline{u}$b, $\underline{u}$a, $\underline{d}$a], $\underline{d}$b] is correct; in it, $\underline{u}$a is "in the scope" of the innermost $\underline{d}$a.
□

Because the translation, into a list of addressing pairs, of an inner block depends on the declarations in the blocks surrounding that block, it is impossible to consider translation of an inner block as a function of that inner block alone. The only thing we can do is to consider translation as a function of both a block and a *context*, where the context represents all that is necessary to translate the block correctly. This can be formalised as follows.

For each block, we consider the list of names declared in that block. For this purpose, we introduce function D, mapping Seq's to lists of names, as follows — where a has type Name and where x and y denote Seq's — :

$$
\begin{aligned}
& D.[\,] && = && [\,] \\
\& \quad & D.(\underline{d}a\,;x) && = && a\,;D.x \\
\& \quad & D.(\underline{u}a\,;x) && = && D.x \\
(0) \quad \& \quad & D.(\underline{s}y\,;x) && = && D.x \quad .
\end{aligned}
$$

The context of a block now is a list of such lists, one for each surrounding block. (To avoid case analysis between local and global variables, we define each block to surround itself.) The translation of a program x now is T.[D.x].x , where function T is defined as follows — in which c denotes the context — :

$$
\begin{aligned}
& T.c.[\,] && = && [\,] \\
\& \quad & T.c.(\underline{d}a;x) && = && T.c.x \\
\& \quad & T.c.(\underline{u}a;x) && = && ap.c.a\,;T.c.x \\
(1) \quad \& \quad & T.c.(\underline{s}y;x) && = && T.(D.y\,;c).y \mathbin{+\!\!+} T.c.x \quad .
\end{aligned}
$$

In this definition, function ap maps contexts and names to address pairs; its definition is irrelevant here. Notice that an inner block y is translated within the "extended context" D.y ; c , as it should be.

The translation of x is T.[D.x].x . One way to evaluate this is to evaluate D.x first, yielding a list d (say), and then to evaluate T.[d].x . Both evaluations involve inspection of program x , which is the reason why this is called a *two-pass* translation process. The problem posed by Swierstra is: can we design a one-pass translator ?

To answer this question we have at least two options. One approach, which we shall not pursue here, is to introduce a function F with specification:

$$F.c.x \;=\; T.(D.x;c).x \;.$$

Then we have $T.[D.x].x = F.[].x$. Now we try to derive a recursive definition for F in which neither T nor D occur anymore. This is somewhat laborious, but it can be done.

Another approach is to calculate in the following way:

$$
\begin{array}{ll}
 & T.[D.x].x \\
= & \quad \{ \text{ abstraction, } [\![\cdots]\!] \text{ denotes a "where-clause" } \} \\
 & T.[d].x \quad [\![\, d = D.x \,]\!] \\
= & \quad \{ \text{ once more abstraction } \} \\
 & t \;[\![\; t = T.[d].x \;\&\; d = D.x \;]\!] \\
= & \quad \{ \text{ tupling } \} \\
 & t \;[\![\; [t,d] = [\,T.[d].x \,,\, D.x\,] \;]\!] \\
= & \quad \{ \text{ introduction of function F, see below } \} \\
 & t \;[\![\; [t,d] = F.[d].x \;]\!] \;.
\end{array}
$$

Notice that the recursion in the definition of $[t,d]$ is a *pseudo recursion*: d only depends on x, not on itself. The last step of this derivation is correct if F satisfies:

(2) $\qquad F.c.x \;=\; [\,T.c.x \,,\, D.x\,] \;.$

Deriving a recursive definition for F is a matter of straightforward calculation; the most complicated case is $F.c.(\underline{s}y;x)$, for which we derive:

$$\begin{aligned}
& F.c.(\underline{s}y ; x) \\
= & \quad \{ (2) \text{ (spec. of } F) \} \\
& [T.c.(\underline{s}y;x) , D.(\underline{s}y;x)] \\
= & \quad \{ (1) \text{ and } (0) \text{ (definitions of } T \text{ and } D) \} \\
& [T.(D.y;c).y \mathbin{++} T.c.x , D.x] \\
= & \quad \{ \text{abstraction and tupling} \} \\
& [T.(D.y;c).y \mathbin{++} tx , dx] \; [\![[tx,dx] = [T.c.x , D.x]]\!] \\
= & \quad \{ \text{idem} \} \\
& [ty \mathbin{++} tx , dx] \\
& [\![[ty,dy] = [T.(dy;c).y , D.y] \;\&\; [tx,dx] = [T.c.x, D.x]]\!] \\
= & \quad \{ (2) \text{ (spec. of } F) \text{ (twice), by induction hypothesis} \} \\
& [ty \mathbin{++} tx , dx] \\
& [\![[ty,dy] = F.(dy;c).y \;\&\; [tx,dx] = F.c.x]\!] .
\end{aligned}$$

Thus we obtain the following definition for F —derivations for the other cases omitted— :

$$\begin{aligned}
& F.c.[] && = [[], []] \\
\& \; & F.c.(\underline{d}a;x) && = [t , a;d] \; [\![[t,d] = F.c.x]\!] \\
\& \; & F.c.(\underline{u}a;x) && = [ap.c.a;t , d] \; [\![[t,d] = F.c.x]\!] \\
\& \; & F.c.(\underline{s}y;x) && = [ty \mathbin{++} tx , dx] \\
& && \quad [\![[ty,dy] = F.(dy;c).y \\
& && \quad \& \; [tx,dx] = F.c.x \\
& && \quad]\!] .
\end{aligned}$$

Furthermore, we have derived —see previous page— :

$$T.[D.x].x = t \; [\![[t,d] = F.[d].x]\!] .$$

The right-hand side of this equality can indeed be considered as a one-pass translator.

In the realm of attribute grammars a distinction is made between *synthesized* and *inherited* attributes. Each synthesized attribute is a function of the (abstract-syntax) tree it belongs to and of the inherited attributes of that tree. For each attribute introduced, the designer of the attribute grammar must decide whether it be a synthesized or an inherited one. Similarly, for each value introduced, the functional programmer must decide whether it is a new function or whether it is added as a new parameter to existing functions. I believe that both design problems are essentially the same; whether their solution is expressed in the formalism of attribute grammars or in functional-program notation is of minor importance. In the case of the translation problem we had no choice: we were forced to introduce the context as an additional parameter of the translation function.

In his Ph.D. thesis M.F. Kuiper claims that attribute grammars can be used to derive efficient functional programs. In view of the above observation this is not so surprising. Until now, however, it is my firm impression that the use of attribute grammars is only a roundabout way to achieve what can also be achieved directly. Here tupling comes in: tupling is a program transformation technique for combining several functions into one, often with the pleasant side-effect of improving the program's efficiency. Notice that tupling can be applied almost mechanically.

Both the examples used by Kuiper to support his claim, in chapter 6 of his thesis, and the examples used by T. Johnsson, in his paper "Attribute grammars as a functional programming paradigm", fail to convince me: I can easily solve these problems, in a calculational way, without the use of attribute grammars. Let me, therefore, conclude with the following conjecture:

> It may very well be that functional programming provides an alternative to attribute grammars, and that the former lends itself better to formal manipulation than the latter. (If you want nondeterminism: replace "functional" by "relational".)

<u>references</u>

T. Johnsson: "Attribute grammars as a functional programming paradigm", in:
G. Kahn (ed.): "Functional programming languages and computer architecture". Springer-Verlag, LNCS 274, 1987.

M.F. Kuiper: "Parallel attribute evaluation". Ph.D. thesis, Rijksuniversiteit Utrecht, 1989.

Eindhoven, 16 july 1990

Rob R. Hoogerwoord
department of mathematics and computing science
Eindhoven University of Technology
P.O. Box 513
5600 MB Eindhoven